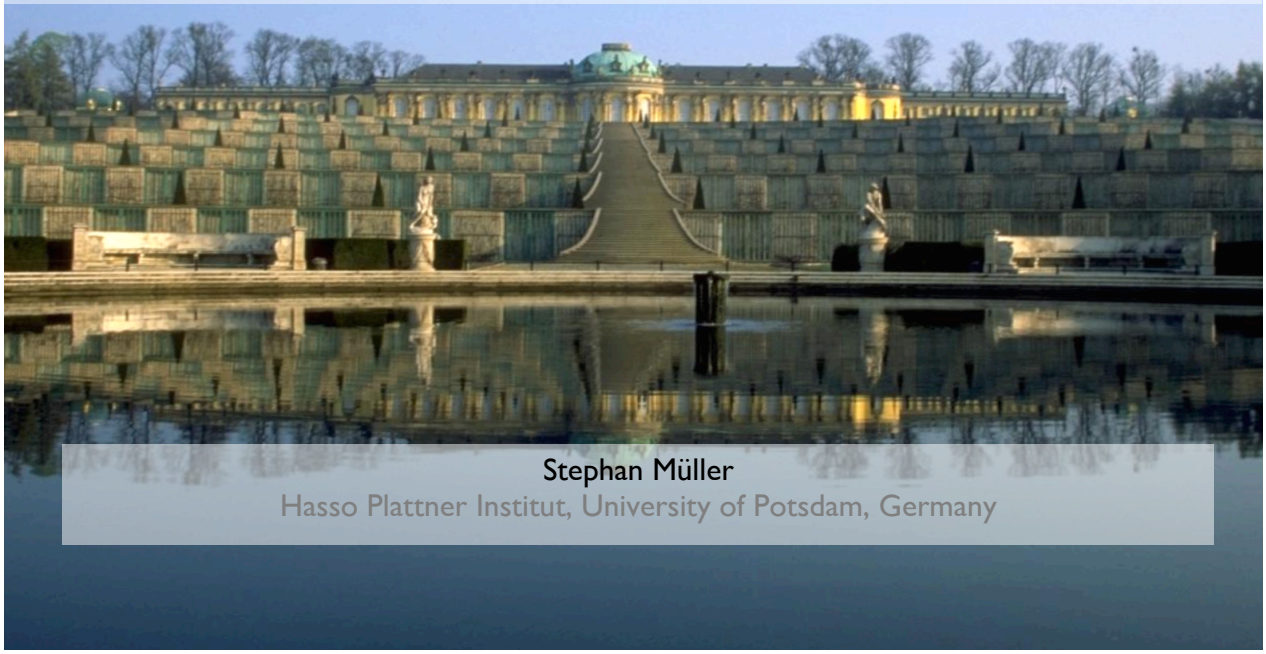
Aggregate Cache



Stephan Müller
Hasso Plattner Institut, University of Potsdam, Germany

Aggregates in Enterprise Applications

- Current handling
  - Application-maintained sum tables in OLTP systems
  - Materialized views in OLAP systems
- Problems
  - Increased application complexity
  - Reduced throughput of insert statements
  - Reporting on stale and redundant data
  - Inflexible analytics due to predefined materialized views
- Goal: highest flexibility by operating on the highest granularity
  - Remove predefined materialized aggregates and sum tables
  - Execution of both transactional AND analytical queries on a single system
  - Reduction of application complexity
  - Flexible, real-time analytics
- Outlook on future enterprise application workloads: increasingly **higher share of aggregate queries**

The Aggregate Cache

- Idea: **cache** the aggregate query result on **main partition** and **combine** it with an **on-thy-fly** aggregation on **delta partition**
  - Supporting distributive or algebraic aggregate functions (SUM, COUNT, AVG, MIN, MAX)
  - One aggregate cache entry per unique combination of table id, grouping attributes, aggregates, and filter predicates
- Materialized aggregate maintenance is delayed (or skipped) until delta merge process
  - Cached aggregate "inconsistency" caused by **inserts** is compensated by on-the-fly aggregation on delta partition
  - Cached aggregate "inconsistency" caused by **updates and deletes** (invalidations in main partition) is compensated by revalidation using bit vectors on main partition & on-the-fly aggregation on delta partition

Example: Aggregate Query

Facts							
Main				Delta			
ID	Date	Prod	Amt	ID	Date	Prod	Amt
1	1/1/2013	1	100				
2	1/1/2013	1	-50				
3	1/1/2013	2	30				
4	1/1/2013	2	60				
5	1/2/2013	1	-10				

```
SELECT Date, Prod, SUM(Amt) FROM Facts GROUP BY Date , Prod
```

Aggregates		
Date	Prod	SUM(Amt)
1/1/2013	1	50
1/2/2013	1	-10
1/1/2013	2	90

Example: Cached Aggregate Query

Facts							
Main				Delta			
ID	Date	Prod	Amt	ID	Date	Prod	Amt
1	1/1/2013	1	100				
2	1/1/2013	1	-50				
3	1/1/2013	2	30				
4	1/1/2013	2	60				
5	1/2/2013	1	-10				
				6	1/2/2013	1	20
				7	1/1/2013	3	50
				8	1/1/2013	3	-10

```
SELECT Date, Prod, SUM(Amt) FROM Facts GROUP BY Date , Prod
```

Aggregates		
Date	Prod	SUM(Amt)
1/1/2013	1	50
1/2/2013	1	-10
1/1/2013	2	90

Aggregates		
Date	Prod	SUM(Amt)
1/2/2013	1	20
1/1/2013	3	40

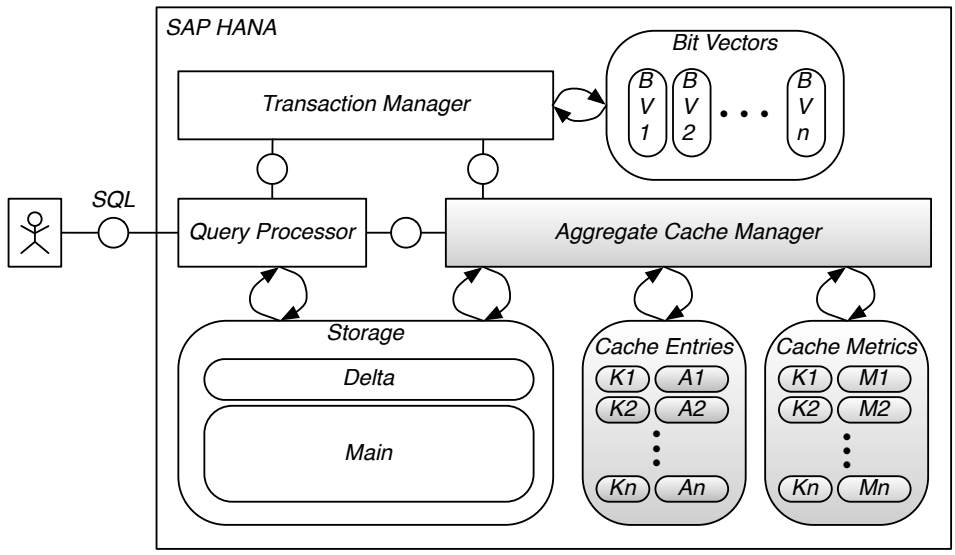
Main (cached)

Delta (on-the-fly)

```
SELECT Date, Prod, SUM(Amt)
FROM (Select * FROM CachedAgg1234 UNION ALL Select * FROM Facts_Delta) GROUP BY
Date , Prod
```

Aggregates		
Date	Prod	SUM(Amt)
1/1/2013	1	50
1/2/2013	1	10
1/1/2013	2	90
1/1/2013	3	40

The Aggregate Cache



Cache Admission and Eviction Strategies

- Goal
  - Limit cache growth and storage requirements
  - Reduce cache overhead and maintenance
- Cost-based approaches
  - Tuple counts
  - Execution times
  - Result set sizes
  - Main invalidations
- Two stages
  - Cache admission and replacement during query execution
  - Benefit analysis during cache maintenance (merge process)
- Results: performance improvements of ~40% over LRU and ~15% over existing cost-based metrics (Watchman/Dynamat)

Deletes and Updates

- Record validity is handled through create and delete bit vectors
  - Delete: set delete bit vector
  - Update: set delete bit vector & insert new value in delta
- Efficient retrieval of invalidated records through bit vectors
- Applying invalidations to the cache entry ensures a consistent result

row_id	Create Baselist	Delete Baselist	Create Chunks	Delete Chunks
1	1	0		
2	1	0	empty	empty
3	1	0	chunk	chunk
4	1	1		
5	1	0		T1
6	1	0	empty	T2
7	1	0	chunk	0
8	1	0		0
9	1	0	0	
10	1	0	0	empty
11	0	0	T2	chunk
12	1	0	0	

1.

	111111111101 (create bit vector)
XOR	000100000000 (delete bit vector)
=	111011111101

2.

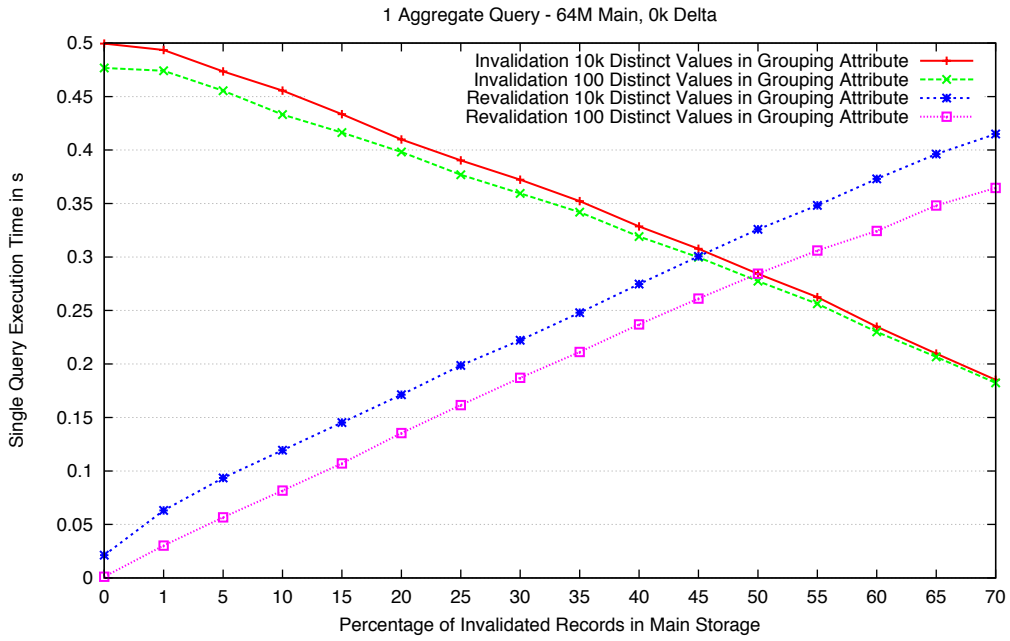
	111011111101
ADD	000000000010 (create chunk)
=	111011111111

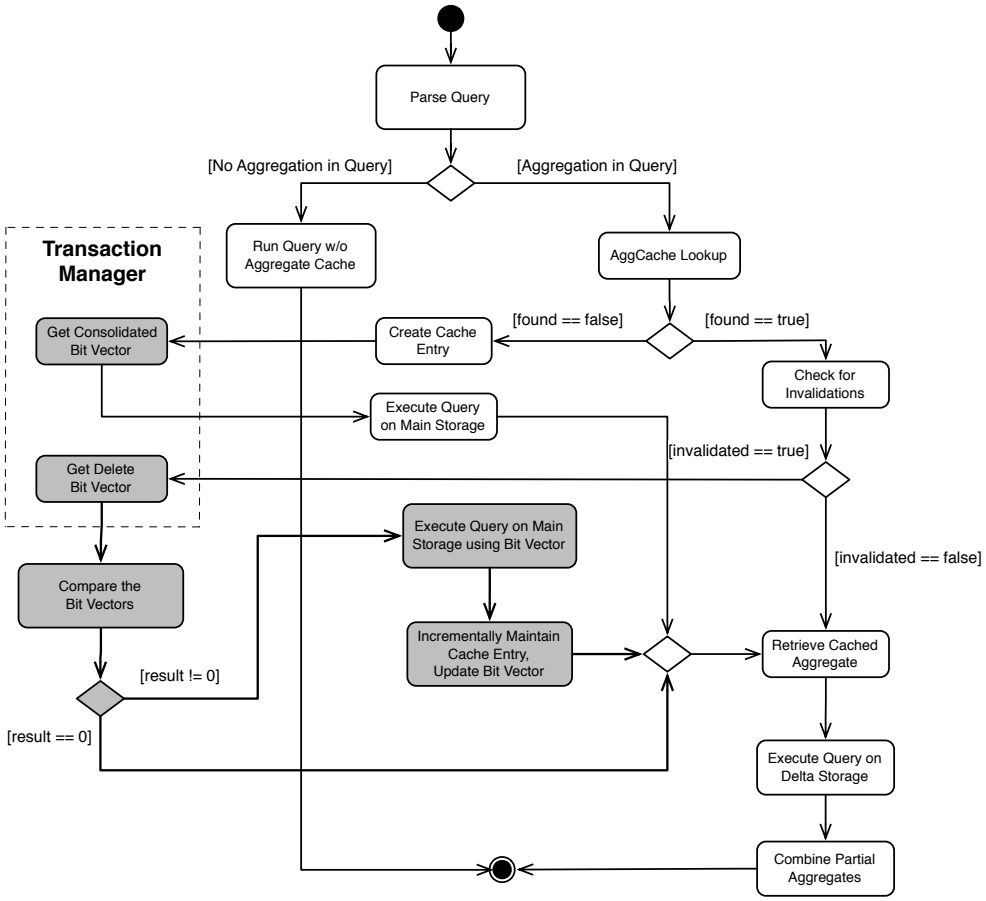
3.

	111011111111
SUB	000001000000 (delete chunk)
=	111010111111

4. **111010111111**

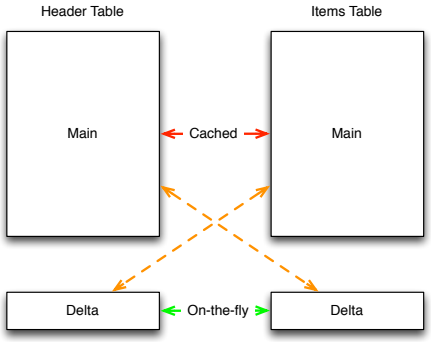
Invalidation Ratio





Aggregates on Multiple Relations

- Increased materialized aggregate maintenance for queries joining multiple (partitioned) tables
- The number of join operations depends on the number of involved tables



- Patterns found in enterprise applications
  - Text and configuration tables are rarely changed (empty deltas)
  - Header and items tuples are created and inserted in a **single transaction**
  - In some cases (adding items to a sales order), items can be added to the header after initial creation
- Idea: add temporal relation to matching tuples during insertion and execute join pruning based on this "**object awareness**" information

time

Main Partition

H_{main}

PK	T_{PK}	Cat
100	1	A
200	2	C
400	4	B

$H_{main} \bowtie I_{main}$ cached (expensive)

I_{main}

PK	T_{PK}	FK	T_{FK}	Amt
31	1	100	1	30
34	2	200	2	50
24	3	200	2	20
35	4	400	4	40

$H_{main} \bowtie I_{delta}$ ✗

$H_{delta} \bowtie I_{main}$ ✗

Delta Partition

H_{delta}

PK	T_{PK}	Cat
150	5	C
250	8	B

$H_{delta} \bowtie I_{delta}$ on-the-fly (cheap)

I_{delta}

PK	T_{PK}	FK	T_{FK}	Amt
52	6	150	5	30
23	8	250	8	50
42	9	250	8	20

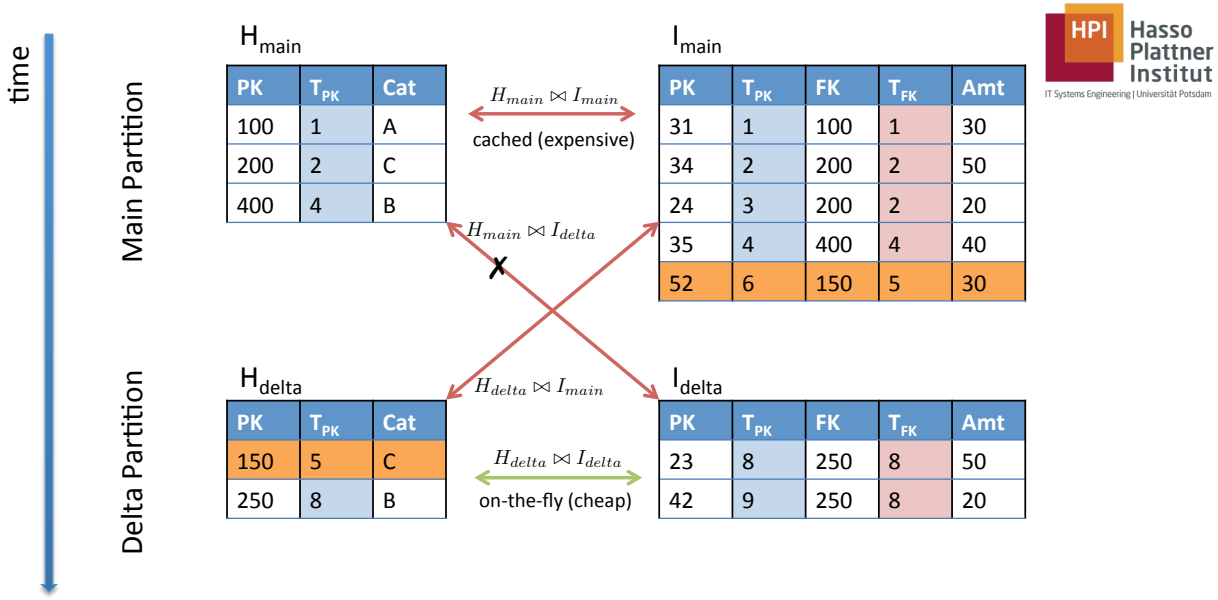
Enforce matching dependency during inserts:

$$\forall i \in I, i[FK] = h[PK] : i[T_{FK}] = h[T_{PK}]$$

Dynamic join pruning during query execution:

$$max(H_{main}[T_{PK}]) < min(I_{delta}[T_{FK}]) \longrightarrow H_{main} \bowtie_{H[PK]=I[FK]} I_{delta} = \emptyset \quad \checkmark \quad 4 < 5$$

$$max(I_{main}[T_{FK}]) < min(H_{delta}[T_{PK}]) \longrightarrow I_{main} \bowtie_{H[PK]=I[FK]} H_{delta} = \emptyset \quad \checkmark \quad 4 < 5$$



Enforce matching dependency during inserts:

$$\forall i \in I, i[FK] = h[PK] : i[T_{FK}] = h[T_{PK}]$$

Dynamic join pruning during query execution:

$$max(H_{main}[T_{PK}]) < min(I_{delta}[T_{FK}]) \longrightarrow H_{main} \bowtie_{H[PK]=I[FK]} I_{delta} = \emptyset$$ ✓ 4 < 8

$$max(I_{main}[T_{FK}]) < min(H_{delta}[T_{PK}]) \longrightarrow I_{main} \bowtie_{H[PK]=I[FK]} H_{delta} = \emptyset$$ ✗ 5 < 5